Los Alamos
NATIONAL LABORATORY

research note

Computer and Computational Sciences

CCS–4:Transport Methods Group

To/MS: Distribution
From/MS: Thomas M. Evans/CCS–4 D409
Todd J. Urbatsch/CCS–4
Phone/FAX: (505)665–3677
Symbol: CCS–4:02-12(U) (LA-UR-02-2445)
Date: 29-APR-2002

Subject: Post—Mortem Review of the Jayenne Code Project (U)

Executive Summary

The JAYENNE project is an Implicit Monte Carlo (IMC) code and research project in CCS–4. The goal of the JAYENNE project is twofold. The first goal is to provide high quality, parallel computational IMC packages to ASCI code projects. The second goal of the project is to investigate new IMC methods and incorporate them into deliverable packages. This paper will focus on progress made toward the first goal, namely, the software packages produced in the JAYENNE project.

To date, the JAYENNE project has produced three IMC packages, MILAGRO, MILSTONE, and WEDGEHOG. MILAGRO is a multi-D, parallel, stand-alone IMC code. MILSTONE and WEDGEHOG are IMC packages that are designed to work with Computational Fluid Dynamics (CFD) applications. In addition to the codes, the JAYENNE project has produced a large amount of documentation. A full bibliography—pertaining to code related aspects of the JAYENNE project—is given in the references to this paper.

The purpose of this paper is to provide material for a comprehensive, internal CCS–4 post-mortem review of the software practices of the JAYENNE project. In the sections that follow we will describe the software engineering practices that we have incorporated in the construction of MILAGRO, MILSTONE, and WEDGEHOG. Furthermore, we will present analysis of the successes and failures of the JAYENNE code projects.

1. Introduction

The JAYENNE project was started on 01-OCT-1997[1]. Its primary mandate was to produce parallel, multi-D IMC packages for ASCI codes. A secondary mandate, in support of the first, was to investigate new methods to improve IMC calculations. The fundamental mandates have remained unchanged since the project's start date. The JAYENNE project team is housed in the Monte Carlo team of the Transport Methods Group and, for the majority of its existence, has consisted of two TSM's, Todd Urbatsch, CCS–4 and Tom Evans, CCS–4.[2] For the second half of FY01, Mike Buksas, upon joining CCS–4, worked directly on the JAYENNE project team, where he improved the

[1]The JAYENNE project is often referred to as the MILAGRO project. Technically, MILAGRO is a code *within* the JAYENNE project.

[2]The Transport Methods Group was originally group XTM, then X–6, until finally becoming CCS–4 on 01-OCT-2000. For convenience and clarity, we will use CCS–4 as the group designation throughout this report.

IMC particle class [4] and added a new IMC particle with hybrid implicit/explicit absorption [3,32]. Therefore, over the 4.5 years since its inception, the JAYENNE project has been allocated a total of 9.5 FTE-years, yielding an average of 2.11 FTEs per year.

Currently, the JAYENNE project contains 3 codes, MILAGRO, WEDGEHOG, and MILSTONE. Each code solves the thermal radiative transfer equations using Monte Carlo transport. The time-Implicit Monte Carlo (IMC) method implemented in these codes was derived by Fleck and Cummings [11]. Details about the codes and software practices employed in their construction are given in § 4.

This note serves as the basis for an internal, CCS–4 post-mortem review of codes within the JAYENNE project. Section 2 describes the goals of the project. In particular, it will answer questions about customers, delivery dates, and project targets. Section 3 illuminates the requirements of the project.

Section 4 describes the details about the project. In this section we will briefly describe the codes within the JAYENNE project. We also describe the Software Quality Engineering (SQE) practices that have been employed by the project. Section 5 chronicles the history of the JAYENNE project from inception to the current day.

Finally, Section 6 will discuss project success and failure points. We will discuss which goals were met, why they were met, and why other goals were not met. Additionally, we will talk about the features of our software engineering process that have worked effectively. We will also discuss shortcomings of the project.

The information presented in this report is summarized from a wide array of documentation produced in the JAYENNE project. A complete bibliography of documentation produced by the project is listed in Refs. [3–9, 18–33].

2. **Project Goals**

This section will answer the following questions:

- What was the project supposed to accomplish?
- Who were the customers?
- When was it supposed to be delivered?

The JAYENNE project was initiated to produce IMC packages for the Accelerated Strategic Computing Initiative (ASCI). As such, the customers for JAYENNE codes are the four ASCI projects, Blanca, Crestone, Antero, and Shavano. Because the ASCI projects have different priorities for physics and numerics, we have established the following delivery order of IMC packages,

1. Crestone
2. Shavano
3. Antero
4. Blanca

Needless to say, priorities changed, and will continue to change, due to the whimsical nature of ASCI program management. Additional customers for JAYENNE code products exist in both X-division and the tri-lab community.

The original delivery date for IMC to the Crestone project was 01-OCT-1998. No dates have been locked down for any period of time after this initial delivery date. Additionally, no formal delivery dates have been set for the other ASCI projects.

3. **Project Requirements**

This section will answer the following questions:

- What were the requirements?
- Who defined/controlled them?
- Were they documented?
- Did they change significantly during the course of the project?

The original requirements for the JAYENNE project were to provide an IMC package with the following capabilities:

- transport on a $3D$, XYZ, orthogonal, adaptive mesh,
- parallelism based on a fully domain decomposed mesh,
- gray and multigroup,
- random walk.

The customers, Crestone in this case, dictated the requirements in concert with the IMC team and CCS–4 group management. Low-level requirements were made by the IMC team. These requirements are

- results are reproducible, regardless of parallel topology,
- codes are developed in C++ that corresponds rigidly to the ANSI standard [13],
- all code must be capable of compilation by the GNU C++ (gcc) compiler (version 3.0.1 or greater),
- all code must be levelized,
- all code must be unit tested,
- all code must be checked for memory leaks.

Low-level, internal requirements undergo periodic changes as we learn more about large-scale, scientific computing.

Originally, all of our requirements for Crestone were documented in Ref. [19]. However, we have realized that our customers run on different time-scales. They were more informal and changed

requirements often and on-the-fly. Our attempts at formalization and documentation were at odds with their style. As we became more cognizant of the users' real needs, we would simply deliver them and announce it with a release note. Nonetheless, we still prefer to produce requirements documents when interacting with a new customer.

The original requirements underwent one significant change since project inception. After our first deliverable, it became clear from discussions with users that $2D$, RZ was more important than $3D$. Thus, we adjusted our requirements to emphasize $2D$, RZ. Additionally, our packages were considered useless without a multigroup capability by some users. This attitude persisted until they discovered that they could match experimental data with gray transport.

We have settled on an informal approach whereby we agree on a set of desired features with "trusted" members of our client community. We formalize these requirements internally. Through this approach we have found that despite loud, political rhetoric that can sometimes damage relationships and reputations, a logical, scientific approach usually dictates requirements at some point.

4. **Project Details**

In this section we describe the codes in the JAYENNE project. Additionally, the following questions will be expounded upon:

- What language was used?
- What platforms/hardware were targeted?
- How were parallel communications handled?
- What were the SQE practices?
- What was the process development model?

In other words, in this section we lay out the software-related details of the JAYENNE code project.

4.1. **Codes**

As mentioned in § 1, the JAYENNE project currently contains three codes, MILAGRO MILSTONE, and WEDGEHOG. All three codes/packages solve the thermal radiative transfer equations using Fleck and Cummings' IMC algorithm. MILAGRO is a stand-alone code. It has the unique capability of running on multiple parallel decompositions: domain replication, where the spatial mesh is replicated on each processor, and domain decomposition, where the mesh is divided among the processors.

MILAGRO physics components are generic and can be used to construct a compact application for parallel performance studies. MILAGRO currently runs with an orthogonal structured, non-uniform mesh in 2-D and 3-D and an RZ-Wedge mesh with Adaptive Mesh Refinement; a tetrahedral mesh is being integrated now by Grady Hughes.

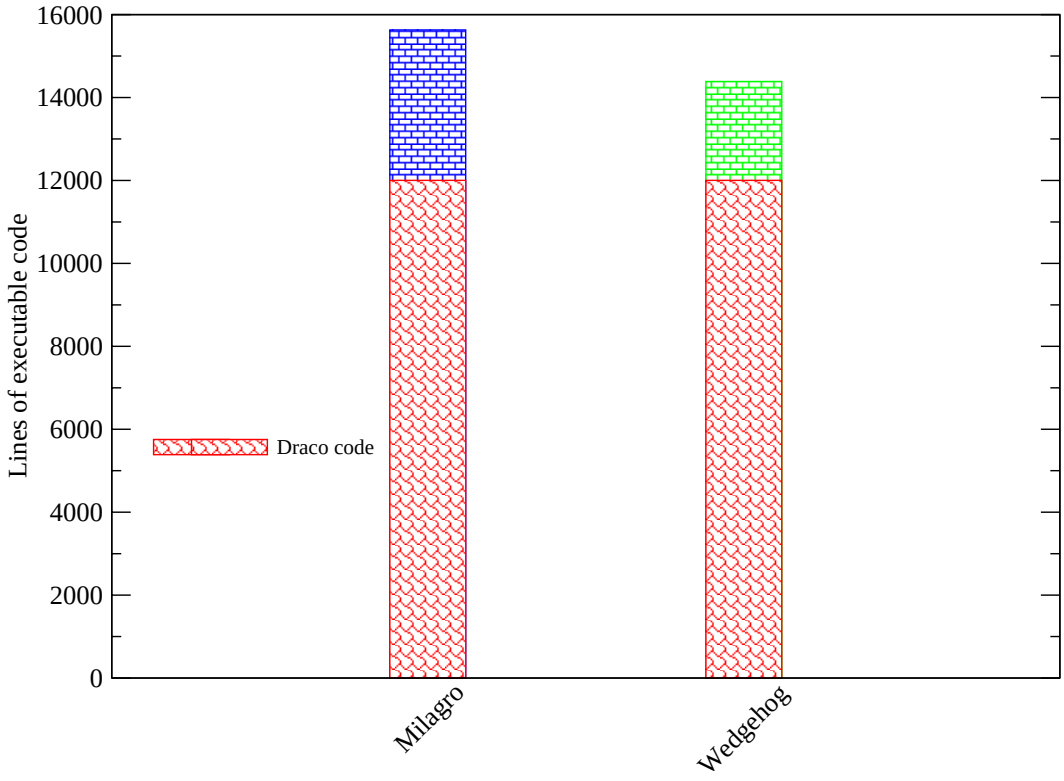

FIG. 1: Code reuse in the JAYENNE project. Here we see that DRACO supplies the majority of code for both the MILAGRO and WEDGEHOG codes.

WEDGEHOG is a $2/3T$, RZ, AMR package that runs in parallel on fully replicated problem domains. MILSTONE is a $2T$, XYZ, orthogonal grid package that runs in parallel on fully decomposed problem domains. All of these codes, MILAGRO, MILSTONE, and WEDGEHOG, are built from a set of core components within the DRACO system [7]. Code components that are common to two or more JAYENNE codes are placed within DRACO. Thus, the JAYENNE packages reuse a substantial portion of common code components. The extent of this reuse is illustrated in Figure 1.

We controlled the broad parallelism strategies through our own codes. We have assumed a distributed memory architecture mainly because that is how our host codes operate. Although the underlying components are capable of domain replication, domain decomposition, and generalized domain replication/decomposition, MILSTONE currently assumes domain decomposition because replication of large meshes is not feasible. WEDGEHOG, on the other hand, utilizes a more efficient domain replication topology because the host codes can afford to replicate the problem data in $2D$ problems. Either of these strategies can be changed with moderate effort because the underlying components support multiple parallel schemes.

Fine grained parallelism in our code is confined to a few places. We use only a subset of message passing functions, blocking send/receives, non-blocking send/receives, barriers, and reductions. Message passing primitives are contained in the DRACO C4 package. As such, our code need not change when C4 uses a new or different communication vendor. However, in all of our development,

C4 has used MPI [12,17]. We intend to explore the usage of OpenMP [16] in the future.

All JAYENNE project codes are built upon an extensive foundation of verification using Design-by-Contract assertions, component tests, and comparison to analytic test problems, all of which are run nightly. These issues are discussed in detail in § 4.2.

Codes in the JAYENNE project utilize Object-Oriented/Generic design principles [1,15] implemented with C++. A key design feature, that is also a requirement of the project, is levelization. We apply the concept of levelization in both the logical and physical designs of the codes. Levelization means that dependencies are never cyclic. From a logical design perspective, levelization requires that *usesA*, *isA*, and *hasA* relationships are all one-sided. In the physical design, levelization requires that if file (directory) A depends on file (directory) B, then B cannot, conversely, depend on A. Figures 2 and 3 show physical levelization diagrams of the component directories in MILAGRO and WEDGEHOG. As described below, levelization is an important part of the JAYENNE SQE process because it allows incremental unit testing.

All JAYENNE codes target ASCI-level computing platforms. In order of importance these are:

1. SGI 0rigin 2000 (Blue Mountain)
2. Compaq ES45 (Q Machine)
3. IBM AIX (ASCI White)

However, we have learned that it is impossible to do day-to-day code development on these platforms. Thus, the primary development platform for the JAYENNE project is Intel and Athlon-based systems running LINUX. We endeavor to write JAYENNE codes as portable as possible (APAP). By conforming rigorously to ANSI standards, we have had little-to-no problems porting the code to multiple platforms. Additionally, we routinely compile and test the code using multiple compilers and profiling tools. These practices, as described in § 4.2, allow us to port the code quite easily.

4.2. **SQE Practices**

We have utilized an extensive set of SQE practices in the JAYENNE project. The software development process that we employ is a variation of the *Unified Process* (UP) combined with *Extreme Programming* (XP) [14]. We have found that an iterative development cycle, as dictated by both the UP and XP, works very well in the ASCI environment. It allows us to respond to changing user needs and requirements quickly and without tremendous disruption.

We utilize a repetitive four stage approach to development consisting of *inception*, *elaboration*, *construction*, and *transition*. In general, our inception stages tend to be very brief and informal. We occasionally make use of *Use Cases*, but, more often, requirements are written down in notebooks and on the whiteboard.

We spend significant time iterating between code and design in the elaboration and construction phases. For example, the initial elaboration stage of the JAYENNE project began with a few months of pencil-and-paper designs, a practice that continues even now alongside code development. The amount of time spent in design has decreased with subsequent elaboration stages due to an evolving

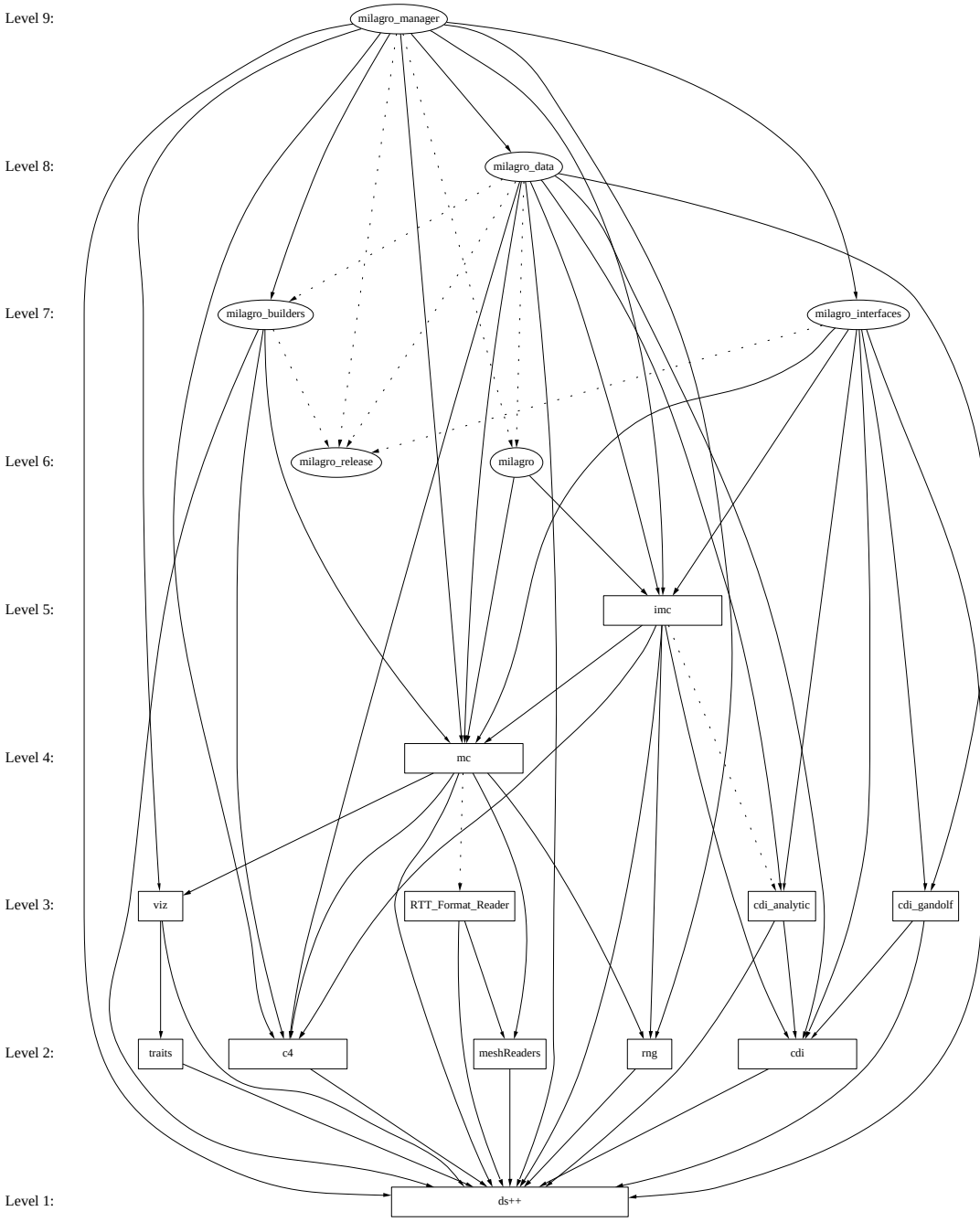

FIG. 2: Physical levelization diagram for the MILAGRO code. Components in boxes are part of the DRACO system. Dotted dependency lines means that the package is required for testing only.

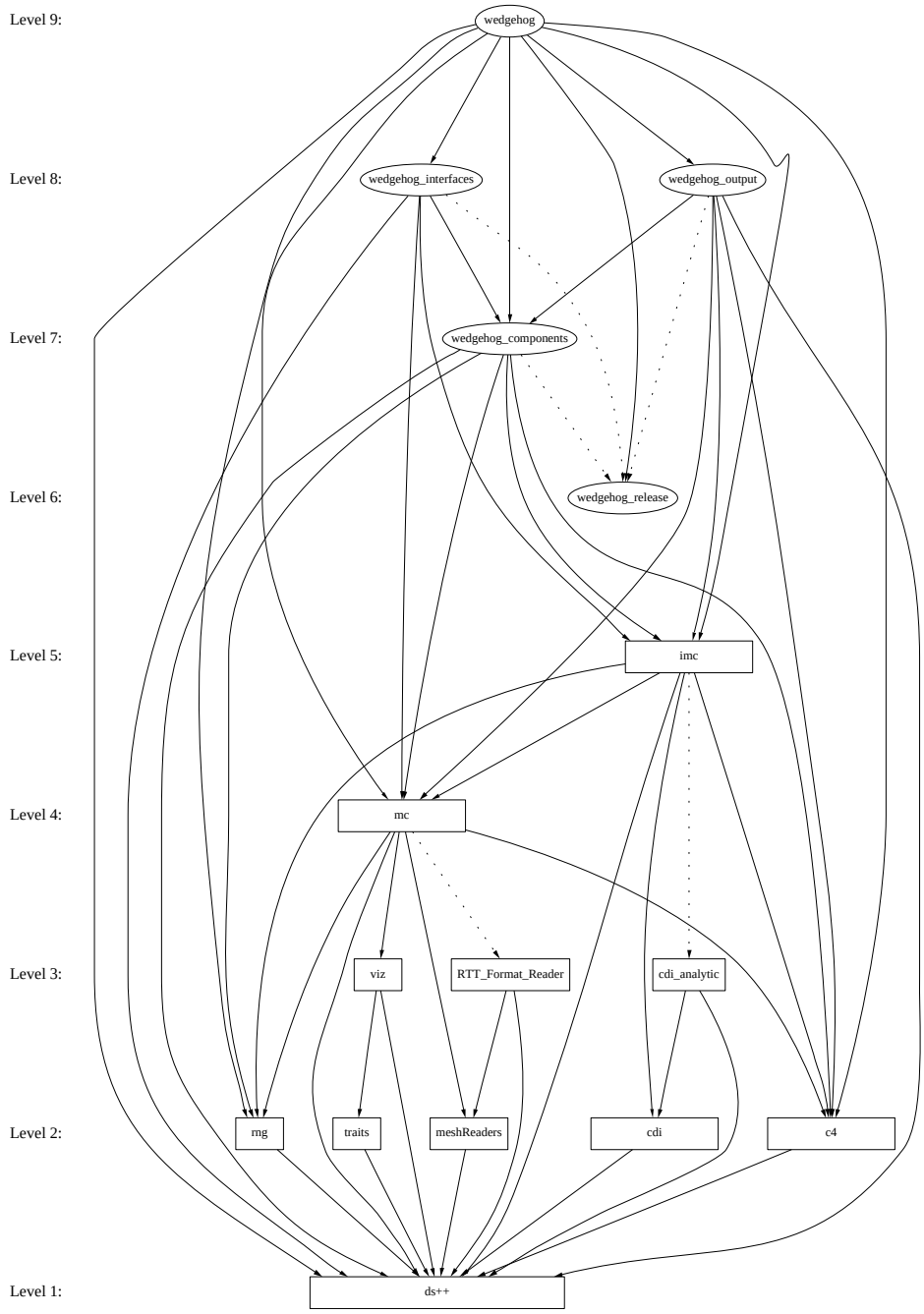

FIG. 3: Physical levelization diagram for the WEDGEHOG code. Components in boxes are part of the DRACO system. Dotted dependency lines means that the package is required for testing only.

maturity in the code design. In all stages of code development, especially the early stages, we rely heavily on UML class and sequence diagrams, level diagrams, and activity charts to illustrate ideas.

An important part of the UP, one which we practice with a high degree of fervor, is the concept of *Staged Delivery*. Our underlying philosophical view toward software delivery is to deliver limited, but relevant, physics capability in software of the highest quality possible. Thus, users do not necessarily get everything they want, but they can be confident that what they do have is correct. This approach has proved successful time after time. An important programmatic benefit of this method of software development is that users can concentrate on validating the physics in the code. They do not have to worry about code correctness.

Refactoring and pair-programming are XP concepts that we have incorporated into the JAYENNE software development process. We have found that our personalities and capabilities are more amenable to pair programming (2 people, 1 keyboard, 1 monitor) instead of code reviews. In general, we feel pair programming is better than code reviews, because it pays to get code right the first time (two heads are better than one, even if one is a cabbage). Pair-programming produces code that is more efficient, more readable, and more likely to be correct. We do, however, utilize code reviews for some simple, straightforward pieces of code.

We have a product-centered approach. We believe it is more important to deliver code in a timely manner than to iterate forever on a piece of code. In other words, we adhere highly to the 80% rule. We do not redesign working code until it becomes necessary. Implicit in the 80% rule is code refactoring. We have found that it is very difficult to determine what parts of a parallel, scientific simulation code are "good enough" until the code is actually used to run real problems. At that time one can make *informed* judgements about what parts of a code need to be refactored and what parts are acceptable as they stand.

Testing is the most developed part of the JAYENNE SQE methodology, and it is the process to which we adhere most rigidly. The JAYENNE testing approach is well documented in Ref. [20]. In summary, we have a multi-layered testing approach. Each C++ class that we develop is matched by a corresponding unit test. The unit tests are written at the same time as the main executable code. The levelization of our physical and logical designs allows us to build these component tests incrementally such that low-level components can confidently be used in higher-level components. At the code executable-level, we run a series of mostly analytical tests ranging from the very simple, such as steady-state infinite medium problems, to the more complex, such as Marshak waves. These tests verify the code physics and inter-component operation. We verify the temporal correctness of the code with automated, nightly runs. More importantly, we can run these test manually to immediately verify the correctness of code changes.

For JAYENNE packages that link to host applications, we produce shunts that "simulate" the actions of the calling host code. This method is applied in the MILSTONE and WEDGEHOG code packages and is documented in Ref. [29]. These shunts take the place of the executable-level tests that are found in MILAGRO. Table 1 shows the number of executable-level tests in MILAGRO, MILSTONE, and WEDGEHOG. For more details on the tests themselves, see Refs. [6, 20, 29].

MILAGRO, in addition to being a research testbed, serves as the baseline code in the JAYENNE project. Because nearly all of the IMC physics components live in DRACO, the output from MILA-

TABLE 1: Number of executable-level tests in MILAGRO, MILSTONE, and WEDGEHOG. The executable-level tests in MILSTONE and WEDGEHOG are actually shunts that simulate the host code.

Code	Number of tests
MILAGRO	218
MILSTONE	5
WEDGEHOG	18

GRO is identical to that produced by WEDGEHOG or MILSTONE. Only the input conditions differ. Thus, we can use MILAGRO to test new code features. When things work acceptably in MILAGRO, we can be confident that they will work in WEDGEHOG and MILSTONE. The ability to perform this type of integration testing is very important because ASCI clients are notoriously difficult to use for verification testing.

An example of the effectiveness of this testing methodology is the 1999 redesign of the parallelism in MILAGRO. We refactored the entire parallel implementation of the code to make it more efficient. These changes had *no* effect on the output of tests because this was an implementation change. Thus, we were able to underwrite the entire parallel design of the code, and, because all of our tests passed at the completion, we were confident that we did not make a mistake that affected the integrity of the code.

Another feature of the JAYENNE project development process is the reliance on Design-by-Contract (DBC) [15]. The combination of levelized component testing, regression testing, and DBC has allowed us to produce virtually bug-free code (one bug in 2 years of use). We are not implying that we write bug-free code; the bugs are simply caught immediately through DBC assertions or component tests. Additionally, DBC checking has allowed us to find several bugs in host codes when our packages are called with incorrect input.

We have attempted to produce significant documentation in the JAYENNE project. Unfortunately, documentation is the item we let slip when deliverable schedules get crunched. Nonetheless, we have produced an extensive array of numbered documentation in Refs. [5–9, 18–31, 33]. In addition to these official documents, we maintain a large library (available in the package/doc directories) of design, activity, and levelization diagrams. We use the UML [2] to document code designs[3]. We do not automatically generate code from UML; the UML is used as a documentation and illustration tool only.

We also do extensive in-code documentation. The tool DOXYGEN is used to convert marked-up comments into HTML, LaTeX, and man pages. This documentation is not intended for a general audience; it is made for code developers.

The combination of unit testing, DBC, and in-code documentation has produced code that is lucid and correct. In fact, the majority of code that we produce in the JAYENNE project is test code, DBC, or comments. Figure 4 shows the breakdown, in lines of code, of executable code, testing code, DBC statements, and comments for the JAYENNE-used parts of DRACO, MILAGRO, and

[3]We use the DIA application to build UML diagrams.

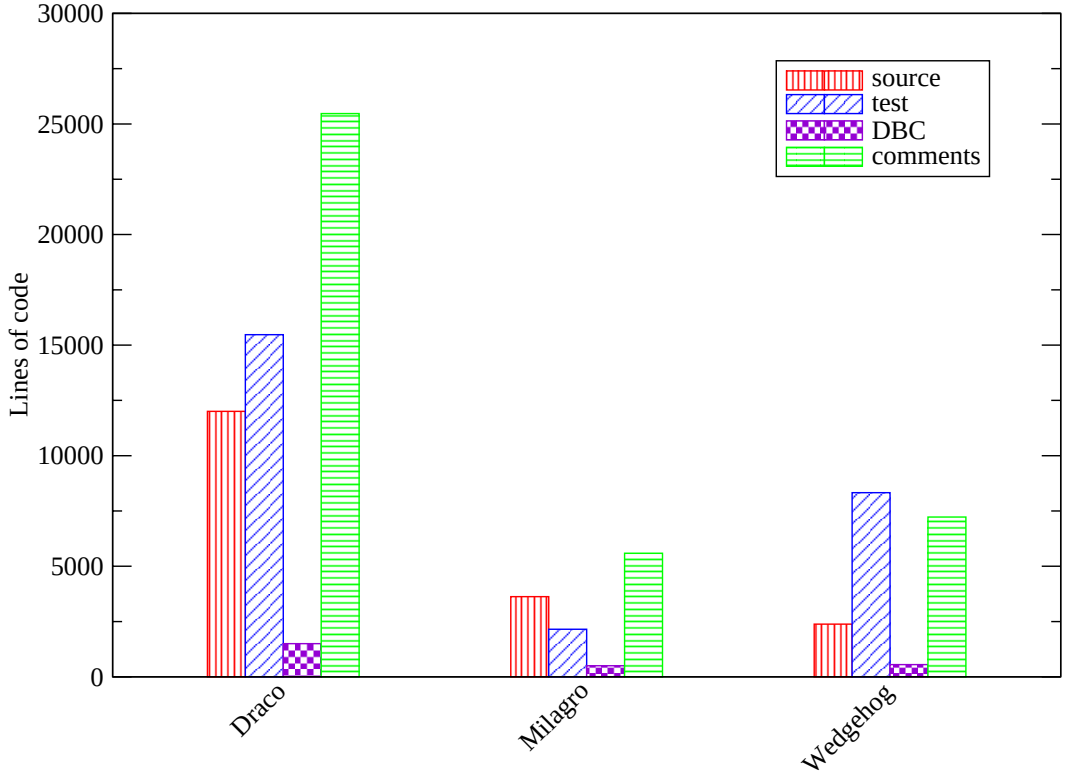

FIG. 4: Breakdown of lines of code in the JAYENNE project. As is shown, more code is written in tests than executable statements.

WEDGEHOG. In each case, the amount of test code exceeds, or is nearly equal to, the lines of executable code. One could argue that this is a dispersion of effort. However, we feel that the degree of confidence attained through this process is well worth the extra time involved in writing tests. Furthermore, this time cost is easily offset when compared against the cost of finding a "buried" bug in a large application, which, experience shows, can take months or even years.

Configuration management in the JAYENNE project is described in Refs. [7, 10]. We use CVS to implement configuration management. Code commits are done daily during the elaboration and construction phases of the code project. During the transition phase, in which we release a version of the code, the code is CVS tagged with a release number. All component and executable-level tests must pass before a release tag is assigned. We maintain pre-built, released versions of the code in the `/usr/projects/jayenne` directory on Blue Mountain and the Q machine. Additionally, HEAD versions of the code are built nightly on the CCS–4 LINUX network as part of the nightly regression testing.

We have not utilized a formal bug-tracking procedure. Currently, bugs are tracked informally, either on the whiteboard, in text files, or in lab notebooks. We have recently begun using the BUGZILLA, web-based bug-tracking system. Also, we have had communication with VMSoftware, the company that produces SOURCEFORGE. However, due to the nature of ASCI users, bug-tracking, in whatever form, is an informal practice. Users announce problems/bugs by email or phone. Thus, any tool that we use is, by practice, confined to intra-group/team usage.

5. **Project History**

This section will ruminate upon the following questions.

- What were the project milestones?
- Were they met?
- What were the effort (FTE) levels?
- What was the personnel turnover?
- Did the involved personnel have the appropriate expertise?
- Did the team personnel work together effectively?
- Did the team and management work together effectively?
- What was the initial estimate of the software size?
- How did the software size (lines of code or feature points) actually evolve?
- How did the bug count evolve as a function of time?
- Was the code extensible enough to handle changing requirements (if any)?

The major deliverables and personal milestones for the JAYENNE project are illustrated in Fig. 5. Starting from scratch on 01-OCT-1997, we delivered a 3-D IMC capability within one year (MILSTONE) (actually 3 weeks less than one year). This deadline was the product of meetings between the IMC team, CCS-4 group management, and ASCI code teams. The delivery was gray, serial, had

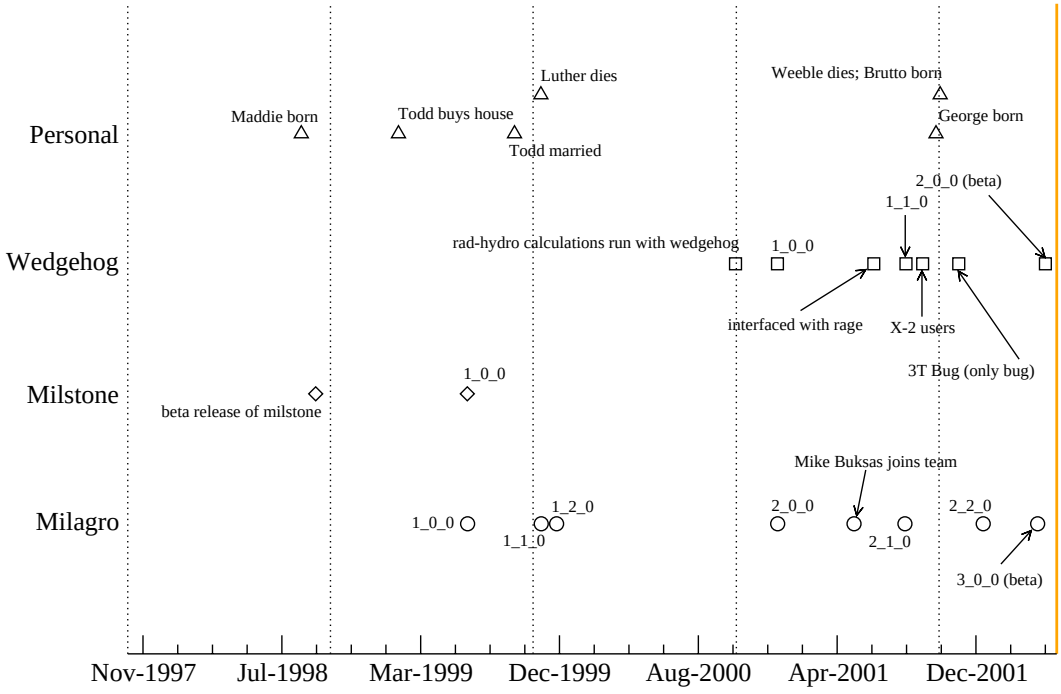

FIG. 5: Milestones, both personal and professional, in the JAYENNE project. Releases are indicated by numbers such as 1_0_0. Dotted lines are fiscal year markers.

no AMR, and lacked momentum deposition. It was "poo-pooed" and deemed useless by prominent users, even though we now know that this capability was adequate to model some experiments. Within 9 months into Fiscal Year 1999, we delivered parallelism using domain decomposition.

Tainted by how our speedy deliveries were received, we retreated to work on increasing the parallel efficiency, improving the energy conservation, and adding momentum deposition, various sources, ratio zoning, restart, AMR, graphics, and a new *RZ* capability, all of which were new, improved, or corrected compared to legacy IMC efforts. We successfully strived to maintain reproducibility while adding all these capabilities.

An *RZ* capability with AMR and momentum deposition (WEDGEHOG) was delivered to a serial host code on 01-OCT-2000, right before we left for a week-long conference. (It ran right out of the box without any problems; the user was amazed and thrilled.) Later, in 2001, another host code picked up WEDGEHOG. Multigroup is being delivered April 2002.

We sincerely feel that the strength of the JAYENNE project lies in the compatibility, both professional and personal (although more often than not, unprofessional), of its team members. We have come to appreciate that the most important factor of a scientific code development team is, like in sports, chemistry. Without the team chemistry that is evident in the JAYENNE project, pair-programming and code reviews would be impossible. An assortment of other activities that have become keystones of our process would be difficult.

The teams interactions with CCS–4 management have been effective. For the most part, CCS–4 management has trusted the team to develop its own process and plans as long as deliverables have been met. The FTE support level has been constant at ~ 2.0 over the course of the project and their has been no personnel turnover.

As mentioned above, there has been one bug that escaped our attention and made it out to the users. The bug was in the WEDGEHOG 3T wrapper, which was used by only one host and one user. We learned that the bug was produced without pair-programming, a practice that may have caught it. Otherwise, all the bugs that we produce are caught by Design-by-Contract assertions, component tests, regression tests, and verification tests—before a delivery is made to the users.

The major change in requirements that has ocurred during the course of the project was a switch in emphasis from $3D$, XYZ to $2D$, RZ. Templating on mesh types allowed us to write and integrate the RZ-Wedge mesh type without changing the physics of the code and to deliver a new capability while retaining most of our verification foundation. Additionally, because we designed the codes to support multiple parallel schemes, we have been able to meet host code parallel requirements without trouble.

6. **Project Success/Failure**

We believe the JAYENNE IMC project is a success. We base this claim on one single, important indicator: customer satisfaction. Our customers are very happy because our packages work as advertised. They are in the refreshing position of not having to debug the tools of their trade. Most importantly, our users avoid building up false intuition because they are free to discuss physics and numerical methods instead of known or unknown code bugs.

Success is built upon failures. We have lost—and will probably continue to lose—political battles. These losses have been a product of many things, such as the immaturity of our project, our immaturity and idealism, our customers' lack of professionalism, and our customers' unwillingness to work as a team. Our reputation hangs in the balance of these political losses and the quality and timeliness of our product. Fortunately, odds have favored the better product.

Success is built on lessons learned. We learned that we need to match the rhythm of our customers without sacrificing scientific integrity or our own mandated requirements. Another lesson learned was in the division of labor. We began our project by steadfastly claiming that we would not work in any of the host codes. We have since learned that our job *is* to work in the host code to interface our packages. Our only hope in that regard is that host code personnel will help with those tasks.

Success is subjective, speculative, and ephemeral. We currently have one major avenue for perceived failure, and that is hooking up to host codes that have substandard quality controls. Because our package is the "new kid on the block," it is usually first in getting blamed for long-standing bugs in the host code. These accusations take an enormous toll on our time, and they adversely and unfairly affect our reputations. We have to learn to effectively deal with these disadvantageous situations, probably by working with the host code to improve their quality controls.

Finally, success is on-going. Given our approach of staged delivery, success, by definition, is followed immediately by failure. Therefore, the life of the JAYENNE project will be a roller coaster of failures

and successes. Hopefully, the failures are short-lived and the successes long-standing.

References

[1] M. H. Austern, *Generic Programming and the STL*. Addison-Wesley Professional Computing Series, Reading, MA: Addison-Wesley, Inc., 1998.

[2] G. Booth, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Object Technology, Reading, MA: Addison-Wesley, Inc., 1999.

[3] M. Buksas, "Low-weight IMC particle termination in Draco," Technical Memo CCS-4:01-33(U), Los Alamos National Laboratory, September 17 2001.

[4] M. Buksas, "Performance improvement in DRACO's IMC particle class," Technical Memo CCS-4:01-19(U), Los Alamos National Laboratory, June 11 2001.

[5] T. M. Evans and T. J. Urbatsch, "MILAGRO: A parallel Implicit Monte Carlo code for 3-d radiative transfer (U)," in *Proceedings of the Nuclear Explosives Code Development Conference*, (Las Vegas, NV), Oct. 1998. LA-UR-98–4722.

[6] T. M. Evans and T. J. Urbatsch, "The Wedgehog Implicit Monte Carlo package." In development, 2001.

[7] T. Evans, "The Draco system for XTM transport code development," Research Note XTM-RN(U)-98–046, Los Alamos National Lab., 1998. LA-UR-98–5562.

[8] T. Evans and R. Roberts, "The draco build system." In development, 1999.

[9] T. Evans, T. Urbatsch, and H.G.Hughes, "IMC C++ class catalog." In development, 2001.

[10] T. Evans, "Draco release policy and procedures," Technical Memo XTM:99–36 (U), Los Alamos National Laboratory, 1999.

[11] J. A. Fleck, Jr. and J. D. Cummings, "An implicit Monte Carlo scheme for calculating time and frequency dependent nonlinear radiation transport," *Journal of Computational Physics*, vol. 8, pp. 313–342, 1971.

[12] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI—The Complete Reference*, vol. 2, The MPI Extensions of *Scientific and Engineering Computation*. Cambridge, MA: The MIT Press, 1998.

[13] ISO/IEC, International Standard, "Programming languages-C++," Tech. Rep. 14882, American National Standard Institute, New York, Sept. 1998.

[14] C. Larman, *Applying UML and Patterns*. Upper Saddle River, NJ: Prentice Hall PTR, second ed., 2002.

[15] B. Meyer, *Object-Oriented Sofware Construction*. Upper Saddle River, NJ: Prentice Hall, second ed., 1997.

[16] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Iinterface," Tech. Rep. 004–2229–001, www.openmp.org, October 1998. Version 1.0.

[17] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI—The Complete Reference*, vol. 1, The MPI Core of *Scientific and Engineering Computation*. Cambridge,

MA: The MIT Press, second ed., 1998.

[18] T. Urbatsch and T. Evans, "New release, Milagro-2_0_0: Details on development and usage." In development, 2001.

[19] T. Urbatsch and T. Evans, "Jayenne data requirements," Technical Memo XTM:98–08 (S), Los Alamos National Laboratory, June 17 1998.

[20] T. Urbatsch and T. Evans, "Regression testing in Milagro," Research Note XTM-RN(U)-99-018, Los Alamos National Laboratory, June 28 1999. LA-UR-99-3482.

[21] T. Urbatsch and T. Evans, "Release notification: Milagro–1_2_0," Research Note X-6:RN(U)–99–037, Los Alamos National Laboratory, November 12 1999. LA–UR–99–6087.

[22] T. Urbatsch and T. Evans, "Release notification: MILAGRO-1_1_0," Research Note X–6:RN(U)-99-033, Los Alamos National Laboratory, October 26 1999. LA-UR-99-5694.

[23] T. Urbatsch and T. Evans, "Momentum deposition in IMC codes," Research Note X-6–RN(U)-00–12, Los Alamos National Laboratory, May 2000. LA-UR-00–2183.

[24] T. Urbatsch and T. M. Evans, "Preliminary scaling results for Milagro," Research Note XTM-RN(U)-99-012, Los Alamos National Laboratory, April 6 1999.

[25] T. Urbatsch and T. M. Evans, "Release notification: MILAGRO-1_0_0," Research Note XTM:RN(U)99-016, Los Alamos National Laboratory, June 4, 1999. LA-UR-2948.

[26] T. Urbatsch and T. M. Evans, "Release notification: Milstone–1_0_0," Research Note XTM:RN(U)–99–017, Los Alamos National Laboratory, June 28 1999. LA–UR–99–3199.

[27] T. J. Urbatsch and T. M. Evans, "The jayenne IMC project plan," Research Note XTM-RN(U)-98-019, Los Alamos National Laboratory, May 1998. LA-UR–98–2262.

[28] T. J. Urbatsch and T. M. Evans, "Strategy for parallel Implicit Monte Carlo," Research Note XTM-RN(U)-98-018, Los Alamos National Laboratory, May 1998. LA-UR–98–2263.

[29] T. J. Urbatsch and T. M. Evans, "Milstone shunt for the marshak 1D problem," Research Note XTM-RN(U)-99-024, Los Alamos National Laboratory, August 6 1999. LA-UR-99-4420.

[30] T. J. Urbatsch and T. M. Evans, "MILAGRO Implicit Monte Carlo: New capabilities and results (U)," in *Proceedings of the Nuclear Explosives Code Development Conference*, (Oakland, CA), Oct. 2000. LA-UR-00-6118.

[31] T. J. Urbatsch and T. M. Evans, "Analytic temperature updates in milagro for T^3 specific heats," Technical Memo CCS-4:01-12(U), Los Alamos National Laboratory, March 12 2001. LA-UR-01-1427.

[32] T. J. Urbatsch and T. M. Evans, "Researching improved low-weight IMC particle termination," Technical Memo CCS-4:01-15(U), Los Alamos National Laboratory, 2001.

[33] T. J. Urbatsch and T. M. Evans, "Reproducibility in parallel Monte Carlo codes," Technical Memo. XTM–99–022 (U), Los Alamos National Laboratory, apr 12 1999. LA–UR–99-1826.

TME:tme

Distribution:
Adams Benjamin T. , CCS–4, D409
Alcouffe Raymond E. , CCS–4, D409
Archuleta Denise G. , CCS–4, D409
Baker Randal S. , CCS–4, D409
Batcho Paul F. , CCS–4, B256
Buksas Michael W. , CCS–4, D409
Carrington David B. , CCS–4, D409
Clark Bradley A. (Brad) , CCS–4, D409
Dahl Jon A. , CCS–4, D409
Evans Thomas M. , CCS–4, D409
Hughes H. Grady, III , CCS–4, D409
Hungerford Aimee L. , CCS–4, D409
Lichtenstein Henry , CCS–4, D409
McGhee John M. , CCS–4, D409
Morel Jim E. , CCS–4, D409
Olson Gordon L. , CCS–4, D409
Pautz Shawn D. , CCS–4, D409
Thompson Kelly G. (K.t.) , CCS–4, D409
Turner Scott A. , CCS–4, D409
Urbatsch Todd J. , CCS–4, D409
Wareing Todd A. , CCS–4, D409
Warsa James S. , CCS–4, D409
CCS–4 D409: